Libra: Contention-Aware GPU Thread Allocation for Data Parallel Training in High Speed Networks

Yunzhuo Liu[1], Bo Jiang[1†], Shizhen Zhao[1], Tao Lin[2], Xinbing Wang[1], Chenghu Zhou[3]
[1]Shanghai Jiao Tong University, [2]Communication University of China, [3]Chinese Academy of Sciences
[1]{liu445126256, bjiang, shizhenzhao, xwang8}@sjtu.edu.cn, [2]lintao@cuc.edu.cn, [3]zhouch@lreis.ac.cn

***Abstract*—Overlapping gradient communication with backward computation is a popular technique to reduce communication cost in the widely adopted data parallel S-SGD training. However, the resource contention between computation and All-Reduce communication in GPU-based training reduces the benefits of overlap. With GPU cluster network evolving from low bandwidth TCP to high speed networks, more GPU resources are required to efficiently utilize the bandwidth, making the contention more noticeable. Existing communication libraries fail to account for such contention when allocating GPU threads and have suboptimal performance. In this paper, we propose to mitigate the contention by balancing the overlapped computation and communication time. We formulate an optimization problem that decides the communication thread allocation to reduce overall backward time. We develop a dynamic programming based near-optimal solution and extend it to co-optimize thread allocation with tensor fusion. We conduct simulated study and real-world experiment using an 8-node GPU cluster with 50Gb RDMA network training four representative DNN models. Results show that our method reduces backward time by 10%-20% compared with Horovod-NCCL, by 6%-13% compared with tensor-fusion-optimization-only methods. Simulation shows that our method achieves the best scalability with a training speedup of 1.2x over the best-performing baseline as we scale up cluster size.**

I. Introduction

Data parallel distributed training with synchronous stochastic gradient descent (S-SGD) has been widely adopted for training deep learning models. In S-SGD, each worker maintains a consistent replica of the model and a split of the dataset. In a training iteration, each worker first computes the gradients based on its own data, and then updates its model replica by aggregating the gradients from all workers, following a predefined communication protocol such as the widely adopted All-Reduce [1–3]. As gradient aggregation can incur time-consuming data communication and may limit the overall training throughput, many recent studies have focused on improving the communication performance [4–7].

Among the numerous software/hardware solutions proposed to mitigate the communication bottleneck of distributed training, overlapping communication behind computation is a primary technique [8–17]. Compared with other solutions like increasing physical bandwidth or gradient compression [18–20], communication-computation overlap neither incurs extra hardware cost nor hinders model accuracy. A typical communication-computation overlap implementation exploits the layered structure of DNN models. As soon as a layer

[†]Bo Jiang is the corresponding author.

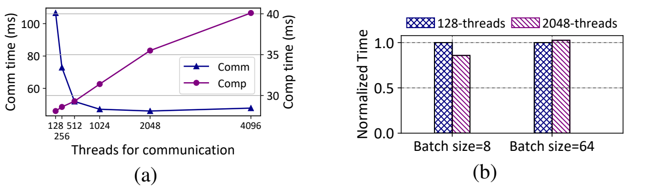

Fig. 1. (a) Time cost of two overlapped communication (112.7 MB) and computation (25.7 ms when executed without overlap) tasks in training Bert-Base on an 8-node V100 GPU cluster inter-connected by 50Gb RDMA. Allocating more GPU threads to communication can decrease the All-Reduce time but increases computation time. (b) Normalized backward time of training ResNet152 using batch size 8 and 64 with two thread allocation policies for the communication task. The time in each group is normalized to the time achieved by the 128-threads allocation.

finishes its gradient computation, the gradient communication of a layer can be performed at the same time as the gradient computation of other layers. Since some layers of a DNN model only have a small number of parameters, launching a communication task for every layer incurs significant startup overhead. As a solution, the tensor fusion technique [10, 11, 21, 22] is used to group layers together to perform communication.

However, the existing computation-communication overlap strategies can be far from optimal. The root cause is that the gradient communication and computation may contend for resources, while the widely-used communication libraries, e.g. NCCL [23] and oneCCL [24], fail to account for such contention when allocating GPU threads to communication. In GPU-based training, All-Reduce communication tasks are launched as GPU kernels and are co-located with gradient computation. Since communication kernels require computation resources to aggregate gradients during All-Reduce and consume corresponding memory bandwidth to move the reduced data to the network, they incur contention with gradient computation that is also in need of these resources [25]. As the network speed of GPU clusters evolves from low-bandwidth TCP to high-speed RDMA, more resources are required by communication kernels in order to saturate the network bandwidth, further exacerbating the resource contention between communication and computation. Fig. 1(a) shows the time cost of two overlapped tasks in training a BERT-Base model on an 8-node GPU cluster with a 50Gb RDMA network. Allocating more GPU threads to communication can decrease the All-Reduce time but noticeably slows down computation, e.g. as we increase the number of threads allocated to communication from 0 to 2048, we observe 38% increase in computation time.

The key to improving communication-computation overlap efficiency is to *balance* the gradient communication time and the gradient computation time with proper GPU-thread allocation. Fig. 1(b) displays the overall backward time (after overlap) of training ResNet152 using two different batch sizes and two communication thread allocation policies. When the batch size is 8 and 128 threads are allocated to the communication task, communication takes longer, and thus increasing the number of communication threads from 128 to 2048 shortens the overall backward time by 14%. In contrast, when the batch size is 64 and 128 threads are allocated to the communication task, computation takes longer, and thus increasing the number of communication threads from 128 to 2048 lengthens the time by 2.6%.

Based on the above observations, our work proposes to mitigate the influence of the contention by adjusting the relative resource allocation of overlapped communication and computation tasks. The contributions of our work are summarized as follows:

- We identify that the thread allocation polices in existing communication libraries are sub-optimal when used for overlapped communication in distributed training. We show through experiment that the influence of resource contention between overlapped communication and computation can be mitigated by balancing their time.
- We formulate an optimization problem that decides the **thread allocation** for each communication task to reduce the overall backward time. We propose a Dynamic Programming (DP) algorithm that finds a near-optimal solution to the optimization problem with polynomial time complexity, and we further extend the algorithm to co-optimize thread allocation with **tensor fusion**.
- We perform both real-world and simulated experiments with popular DNN models to evaluate our approach. The real-world experiment on an 8-node GPU cluster with 50Gb RDMA network shows that our method reduces the backward time by 10% to 20% compared with the widely adopted state-of-the-art distributed training framework Horovod-NCCL, and by 6% to 13% compared with state-of-the-art tensor-fusion-optimization-only methods. Simulation results show that our approach achieves the best system scalability, with a training speedup up to 1.2x over the best-performing baseline.

II. Background

A. Synchronous SGD

SGD. Stochastic gradient descent (SGD) is a widely adopted algorithm for training deep learning models. SGD minimizes the loss function by iteratively updating the model parameters using gradients estimated on mini-batches of training data. More specifically, the updating rule is

$$w_{i+1} = w_i - \lambda \nabla \ell(w_i; X_i), \tag{1}$$

where w_i denotes the model parameters in the i-th iteration, $\nabla \ell(w_i; X_i)$ is the gradient of the loss function ℓ with respect to a mini-batch X_i of the data, and λ is the learning rate.

Synchronous Data-Parallel SGD (S-SGD). S-SGD accelerates the training process by using multiple workers, i.e. processors such as GPUs, to calculate gradients in parallel. The training data is split into N parts and distributed among N workers. In the i-th training iteration, worker n calculates the gradient $\nabla \ell(w_i, X_i^n)$ on a mini-batch X_i^n of its own data. The workers then communicate with each other to aggregates the gradients and update the model according to

$$w_{i+1} = w_i - \frac{\lambda}{N} \sum_{n=1}^{N} \nabla \ell(w_i; X_i^n). \tag{2}$$

Since the workers always use the same model parameters in every iteration, S-SGD is equivalent to non-distributed SGD with mini-batch X_i being the union of X_i^n, $n = 1, 2, \ldots, N$.

B. All-Reduce

Gradient aggregation in S-SGD is typically accomplished by a collective communication algorithm. There are different types of such algorithms [1–3], among which *All-Reduce* is by far the most widely adopted in S-SGD. In All-Reduce each worker receives a partial sum of gradients from another worker, adds to it the local gradients, and passes the result to the next worker. This is repeated multiple times, at the end of which every worker has the sum of all gradients. The sum operations require compute resources provided by GPU threads. The number of allocated threads determines the rate at which gradients are summed and fed to the network. Without enough threads, the network bandwidth may be underutilized.

The total time of a full All-Reduce operation is given by

$$t_{allreduce} = a + \frac{\hat{c}}{x}, \tag{3}$$

where the first term is the startup time and the second the transmission time [10, 11]. Here $\hat{c}$ is the total size of transmitted data, and x the transmission rate, which is affected by the communication thread allocation r. The quantities a and $\hat{c}$ depend on the specifics of the All-Reduce algorithm. In the widely adopted ring-allreduce [1], N workers form a directed ring topology and perform $2(N-1)$ rounds of communication. In each round a worker sends to its next-hop peer $\frac{1}{N}$ gradient size of data. In this case, a and $\hat{c}$ are given by

$$a = 2(N-1) \cdot \alpha, \quad \hat{c} = \frac{2(N-1)}{N} \cdot c, \tag{4}$$

where α is the average one-hop network latency and c is the gradient size.

C. Communication and Backward Computation Overlap

DNN models typically have layered structures. As shown in Fig. 2, in each iteration, forward computation is first performed from layer 1 to layer L to calculate the loss. Backward computation, i.e. backpropagation then calculates the gradients from layer L to layer 1. Gradients for each layer are aggregated through communication and used to update the model. As there is no dependency between the gradient communication of layer $i+1$ and the backward computation of layer i, they can be performed simultaneously once the backward computation of layer $i+1$ completes. This is often exploited to overlap communication and backward computation, effectively hiding part of the communication time.

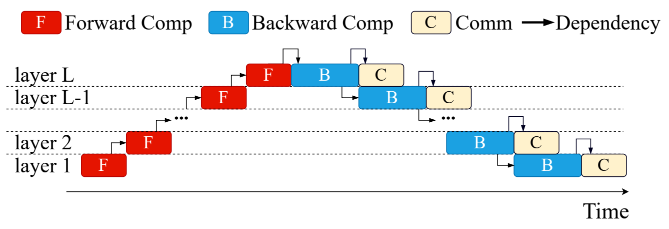

Fig. 2. Communication and computation overlap in DNN training.

D. *Tensor Fusion*

Due to the startup overhead a, launching an All-Reduce for a layer with small gradient size c can be of low efficiency. Tensor fusion technique groups several layers and launches one All-Reduce once their gradients have all been generated. In such way, tensor fusion reduces the total overhead caused by a. Tensor fusion incurs design tradeoffs, i.e. smaller tensor groups cause higher overhead but provide opportunity for making more fined-grained scheduling decisions. Our work solves the problem of co-optimizing tensor fusion with communication thread allocation to reduce overall backward time.

III. Problem Formulation

In this section, we first model the effect of the contention and then we give the problem formulation.

A. *Contention Model*

In GPU-based training with All-Reduce for gradient aggregation, each communication/computation task is launched as a *CUDA kernel* with a user-specified number of GPU threads. Concurrent kernels may contend for resources as their threads are all scheduled to shared hardware compute units called *streaming multiprocessors* for execution. The scheduler is typically implemented in hardware, the details of which remain unknown to users [26, 27]. To capture the contention between communication and computation kernels, we build a model by taking the phenomenological approach rather than from first principles. In our later optimization of backward time, we decide the communication thread allocation to control the compute resource for communication and adjust the compute resource for gradient computation indirectly. Such design avoids modification to the computation libraries in deep learning frameworks and thus reduces deployment cost. Corresponding to this design, we focus on how to empirically measure the relationship between the *time cost of a fully-overlapped communication/computation task* and the *number of threads allocated to the communication kernel*. We collect and analyze overlap examples by training a variety of deep models [28–31] on GPUs of both *Pascal* [32] and *Volta* [33] architectures with random thread allocation policies. Our empirical observations and the models that we find to approximate the relationships are as follows.

Observation 1. *The actual compute resources scheduled to computation decrease approximately linearly with the number of threads r allocated to communication.* The computation time can be approximated by

$$g(b, r) = \frac{b}{\alpha_1 - \alpha_2 \cdot r}, \tag{5}$$

where b is the size of the computation as measured by its execution time without any overlap, and α_1 and α_2 are parameters to determined through curve fitting.

Observation 2. *The transmission rate, i.e. x in Eq.* (3)*, demonstrates a pattern of diminishing returns with the increase of r.* Given r and gradient size c, the transmission time can be approximated by

$$p(c, r) = \frac{\hat{c}}{\gamma_1 - \gamma_2 \cdot \exp(-\gamma_3 \cdot r^{\gamma_4})}, \tag{6}$$

where γ_1, γ_2 and γ_3 are parameters to be determined by curve fitting, and we set an empirical value for γ_4, i.e. 0.6. The quantity $\hat{c}$ is the total size of transmitted data determined by gradient size c and the specifics of the All-Reduce algorithm, e.g. Eq. (4) for ring-allreduce. We use $q(c, r)$ to approximate the transmission time of non-overlapped communication. $q(c, r)$ has the same form as $p(c, r)$ but with a different set of parameters. We fit $p(c, r)$ and $q(c, r)$ with data of overlapped and non-overlapped communication cases respectively.

Note that the functional relationships given by $g(b, r)$ and $p(c, r)$ apply to different models, but each model requires its own set of parameters. Although the effect of the contention between communication and different computation kernels within a model can be different, we do not further distinguish them as we believe such differences can be mostly averaged out at the model level. We will show in Section V-B that our empirical models (5) and (6) achieve a high accuracy when used to predict the backward time. Note that $g(b, r)$ and $p(c, r)$ are regarded as black box functions in our formulation (Sec III-B) and dynamic programming based solution (Sec IV), they can be easily replaced if the functional relationship changes with the development of GPU in the future.

B. *Backward Time Optimization*

Consider a model divided into K groups by tensor fusion. Each group introduces a computation task and a communication task for computing and aggregating gradients, respectively. As shown in Fig. 3, tensor groups are indexed from 1 to K basing on their order in the backward computation. Assume the scheduler is work-conserving for computation, so there is no idle period between consecutive computation tasks. We also assume the communication tasks are launched sequentially. The communication of a group starts once all of its gradients have been generated and the communication of the preceding group has completed. Note that we do not use simultaneous

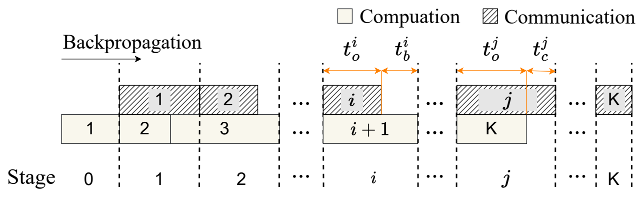

Fig. 3. An example of backward time breakdown. Index on each communication/computation block indicates the tensor group it corresponds to.

communication [11], as the goal is not to maximize network utilization, and we can increase the number of communication threads for that purpose if needed.

The backward process is divided into $K+1$ stages by the start times of the communication tasks. The duration t^i of stage i can be decomposed as $t^i = t_o^i + t_b^i + t_c^i$, where t_o^i is the time for overlapped computation and communication, while t_b^i and t_c^i are the residual, i.e. non-overlapped computation time and communication time, respectively. The subscripts o, b and c stand for overlap, backward computation and communication, respectively. Note that $t_o^0 = t_o^K = 0$, as stage 0 comprises only the computation task of the group 1, while stage K comprises only the communication task of the group K. Also note that t_b^i and t_c^i cannot both be nonzero. In particular, t_b^i is nonzero only if the communication of group i completes before the computation of group $i+1$, whereas t_c^i is nonzero only if the communication of group i completes after the computation of the last group K.

Let $R = \{R_1, R_2, ..., R_J\}$ be the set of possible GPU thread allocations, and let $r_i \in R$ be the thread allocation for the communication of tensor group i. The goal is to choose the right (r_i) to minimize the overall backward time, i.e.

$$\begin{aligned} \min \quad & \sum_{i=0}^{K} (t_o^i + t_b^i + t_c^i) \\ \text{s.t.} \quad & r_i \in R, \qquad\qquad 1 \le i \le K. \end{aligned} \tag{7}$$

Before the calculation of t_o^i, t_b^i, and t_c^i, we first introduce our models for overlap time and residual time when a communication task of size c overlaps a computation task of size b, with r threads allocated to communication.

Overlap Time. The overlap time $f_o(c,b,r)$ is given by

$$f_o(c,b,r) = \begin{cases} 0, & c = 0 \text{ or } b = 0, \\ \min\{p(c,r) + a, g(b,r)\}, & \text{otherwise}, \end{cases} \tag{8}$$

where $p(c,r)$ is the transmission time when communication is fully overlapped, $g(b,r)$ is the computation time when computation is fully overlapped, both given in the previous subsection, and a is the startup overhead in All-Reduce. The smaller one of $p(c,r)+a$ and $g(b,r)$ decides the overlap time.

Residual Time. To determine the residual time, let δ_b and δ_c be the fractions of the remaining work in the computation and communication tasks, respectively, when the overlapped part is excluded. Assuming the processing speeds of both tasks are uniform in time, we have

$$\delta_b = \begin{cases} 0, & b = 0, \\ 1 - \frac{f_o(c,b,r)}{g(b,r)}, & \text{otherwise}. \end{cases} \tag{9}$$

and

$$\delta_c = \begin{cases} 0, & c = 0, \\ 1 - \frac{f_o(c,b,r)}{p(c,r)+a}, & \text{otherwise}; \end{cases} \tag{10}$$

Recall that $q(c,r)$ is the transmission time of a non-overlapped communication task. Assuming uniform processing speed, the residual computation time $f_b(c,b,r)$ and communication time $f_c(c,b,r)$ are then given by

$$f_b(c,b,r) = b \cdot \delta_b, \quad f_c(c,b,r) = (q(c,r) + a) \cdot \delta_c. \tag{11}$$

Calculation of t_o^i, t_b^i and t_c^i. We use s_b^i to represent the size of the overall unfinished backward computation at the beginning of stage i, where $s_b^0 = \sum_{i=1}^K b_i$ and $s_b^K = 0$. As t_o^i and t_c^i are the overlapped and residual time of the communication of tensor group i when it overlaps the remaining computation s_b^i, by Eq. (8) and Eq. (11),

$$t_o^i = f_o(c_i, s_b^i, r_i), \quad t_c^i = f_c(c_i, s_b^i, r_i), \tag{12}$$

where $t_o^0 = t_c^0 = 0$.

As t_b^i is a period of non-overlapped computation, it can be calculated by the difference of the remaining backward computation size, i.e. t_b^i is given by

$$t_b^i = \max\{0, \hat{s}_b^i - \sum_{j=i+2}^{K} b_j\}, \tag{13}$$

where $\hat{s}_b^i$ is the remaining size when communication of tensor group i ends, and $\sum_{j=i+2}^{K} b_j$ is the remaining size when backward computation of tensor group $i+1$ ends. By Eq. (11),

$$\hat{s}_b^i = f_b(c_i, s_b^i, r_i). \tag{14}$$

It is the residual computation time when communication of tensor group i overlaps the remaining computation s_b^i. Finally, we update the s_b^{i+1} as

$$s_b^{i+1} = \hat{s}_b^i - t_b^i. \tag{15}$$

IV. Problem Solution

In this section, we first propose a dynamic programming based method to solve the optimization problem. Our algorithm finds a near-optimal solution with polynomial time complexity. Next, we extend the algorithm to co-optimize thread allocation and tensor fusion.

A. Dynamic Programming Solution

Let $T(i, s_p)$ denote the optimal time when stage i ends, with backward computation of size s_p processed. The optimal backward time is represented by $T(K, \sum_{i=1}^K b_i)$. We use the following equation to compute $T(i, s_p)$ recursively,

$$T(i, s_p) = \min_{1 \le j \le J, n(i) \le s'_p \le s_p} \{T(i-1, s'_p) + h(c_i, s_p, s'_p, R_j)\}. \tag{16}$$

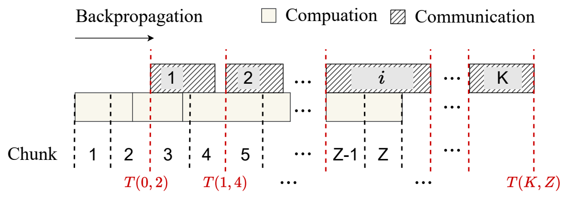

Fig. 4. Discretization of Backward Computation.

The first term $T(i-1, s'_p)$ is the optimal time when stage $i-1$ ends, with backward computation of size s'_p processed. The second term $h(c_i, s_p, s'_p, R_j)$ is the stage time, which is the time of overlapping the communication of tensor group i, whose size is c_i, with the computation of size $s_p - s'_p$, with R_j threads allocated to communication. We use $n(i)$ to denote the total computation size of tensor groups 1 to i. The condition $s'_p \geq n(i)$ ensures that the gradients of tensor group i have been generated when its communication starts at $T(i-1, s'_p)$. The optimal time $T(i, s_p)$ is then obtained by minimizing over all possibilities for s'_p and R_j.

As s_p is continuous, searching every possible s_p is infeasible. Therefore, we discretize s_p by dividing the overall backward computation evenly into Z chunks, with each chunk having a computation size $w = \frac{1}{Z}\sum_{i=1}^{K} b_i$, as shown in Fig. 4. Furthermore, we assume that overlapped communication tasks can start only at the edge of a computation chunk. With abuse of notation, we use $T(i, z)$ to denote the optimal time when stage i ends, with z computation chunks processed. We use the following equation to compute $T(i, z)$ recursively,

$$T(i,z) = \min_{1\leq j\leq J, n(i)\leq z'w\leq zw} \{T(i-1, z') + h(c_i, zw, z'w, R_j)\}. \tag{17}$$

The subproblem is finding $T(i-1, z')$, and we minimize over all possibilities for z' and R_j to get $T(i, z)$. Recall that stage time t^i, i.e. $h(c_i, zw, z'w, R_j)$ in this case, can be decomposed as overlapped time t^i_o, residual computation time t^i_b and residual communication time t^i_c, by Eq. (8) and Eq. (11),

$$\begin{aligned} t^i_o &= f_o(c_i, (z-z')w, R_j), \\ t^i_b &= f_b(c_i, (z-z')w, R_j), \\ t^i_c &= f_c(c_i, (z-z')w, R_j). \end{aligned} \tag{18}$$

The stage time $h(c_i, zw, z'w, R_j)$ is given by

$$h(c_i, zw, z'w, R_j) = \begin{cases} \infty, & z < Z \text{ and } t^i_c > 0, \\ t^i_o + t^i_b + t^i_c, & \text{otherwise.} \end{cases} \tag{19}$$

When $z < Z$, the backward computation of the last tensor group K is not yet completed when stage i ends at $T(i, z)$, so the residual communication time t^i_c must be zero. The first case outputs an infinitely large stage time simply to exclude the possibility violating this condition, i.e. $t^i_c > 0$, when we search all possibilities for z'.

For a given Z, we can find $T(K, Z)$ and the corresponding thread allocations (r^*_i) using dynamic programming with a polynomial time complexity and polynomial space complexity of $O(K \cdot Z^2 \cdot J)$ and $O(K \cdot Z)$ respectively. In practice, we will remove the constraint that each communication starts at the edge of a computation chunk. We use $\hat{T}(K, Z)$ to denote the backward time achieved by (r^*_i) without the constraint. Theorem 1 below shows that with a proper configuration for Z, our algorithm can find a solution that is close to the optimal thread allocation chosen from the set R. Theorem 1 is based on the following three assumptions.

Assumption 1. *Computations in a DNN model are homogenous when contending with communications.*

Assumption 2. *Overlap with computation does not accelerate the execution of communication.*

Assumption 3. *Overlapped execution time does not exceed the sum of sequential execution time.*

The first assumption is based on our study in Sec. III-A, where we find that the time of different overlapped computation tasks in a DNN model can be well-approximated using the same set of parameters in Eq. (5). The second and third assumptions are intuitive and consistent with our observations.

Theorem 1. *Under the above assumptions, the solution (r^*_i) is a $\frac{K-1}{Z}$-approximation to the thread allocation problem (7), i.e.*

$$\hat{T}(K,Z) \leq (1 + \frac{K-1}{Z})T^*, \tag{20}$$

where T^* is the optimal backward time.

The proof is given in Sec. IV-C. Eq. (20) shows that we can choose $Z \geq \frac{K-1}{\epsilon}$ to guarantee that the solution we find achieves a backward time within $1+\epsilon$ times the optimal T^*.

To sum up, for a model with a given tensor fusion plan, our allocation algorithm finds a solution close to the optimal thread allocation in the given set R. This algorithm can be used in combination with existing tensor fusion algorithms [9, 10]. Next, we further extend the DP algorithm to jointly optimize thread allocation and tensor fusion. Co-optimization with tensor fusion provides the opportunity to further balance the communication startup overhead and the benefits of performing more fine-grained communication thread allocation.

B. Thread Allocation and Tensor Fusion Co-optimization

1) Dynamic Programming: The DP algorithm for co-optimization is similar to that in Sec. IV-A. To emphasize the similarity, we abuse the notation and use $T(l, z)$ to denote the optimal time when we merge the first l layers into a tensor group and overlaps their communication with z computation chunks. The optimal backward time is represented by $T(L, Z)$. We use the following equation to compute $T(l, z)$ recursively,

$$T(l,z) = \min_{1\leq j\leq J, n(l)\leq z'w\leq zw, 0\leq l'\leq l} \{T(l', z') + h(\sum_{i=l'+1}^{l} c_i, zw, z'w, R_j)\}. \tag{21}$$

The first term $T(l', z')$ is the optimal time of merging the first l' layers into tensor groups and overlapping their communication with z' computation chunks. We use $n(l)$ to denote the total computation size from layer 1 to layer l. Layer $l'+1$ to l form the new tensor group and its communication overlaps with $(z-z')w$ computation chunks. We search all possibilities for l', z' and R_j to get the minimal time for $T(l, z)$.

For a given Z, we can find $T(L, Z)$ and the corresponding thread allocations $(\hat{r}_i^*)$ using dynamic programming with a polynomial time complexity and polynomial space complexity of $O(L^2 \cdot Z^2 \cdot J)$ and $O(L \cdot Z)$ respectively. We use $\hat{T}(L, Z)$ to denote the backward time achieved by $(\hat{r}_i^*)$ without the computation chunk edge constraint. Theorem 2 below shows that with a proper configuration for Z, our algorithm can find a solution close to the optimal thread allocation chosen from the set R. The proof is similar to that of Theorem 1 and omitted.

Theorem 2. *Under Assumptions 1 to 3, the solution $(\hat{r}_i^*)$ is a $\frac{L-1}{Z}$-approximation to the thread allocation and tensor fusion co-optimization problem, i.e.*

$$\hat{T}(L, Z) \leq (1 + \frac{L-1}{Z})T^*. \tag{22}$$

2) Limited Number of Tensor Groups: Recall that the time complexity of solving $T(L, Z)$ is $O(L^2 \cdot Z^2 \cdot J)$, and L can have a large value as some models may have up to thousands of layers. Theorem 2 suggests that we may need Z to be at the same or a higher order of L to guarantee a solution close to the optimal, such solution can incur high cost. To reduce the solution time, we further add a limit for the maximum number of tensor groups K by explicitly specifying the number of tensor groups of each subproblem in the DP, i.e. with abuse of notation, we use $T(i, l, z)$ to denote the optimal time when we merge the first l layers into i tensor groups and overlaps their communication with z computation chunks. The optimal backward time is represented by $T(K, L, Z)$. We use the following equation to compute $T(i, l, z)$ recursively,

$$\begin{aligned} T(i,l,z) = \min_{1\leq j\leq J, n(l)\leq z'w\leq zw, 0\leq l'\leq l} &\{T(i-1,l',z') \\ &+ h(\sum_{i=l'+1}^{l} c_i, zw, z'w, R_j)\}. \end{aligned} \tag{23}$$

The choice of K is a trade-off. A larger value of K increases solution time but may generate better performance. The implementation of our DP algorithm is given in Algorithm 1 and the complete procedure of LIBRA in Algorithm 2. Note that the model profiling and overlap data collection in Libra (line 1 and line 2 in Algorithm 2) incur only minute-level cost, as we only need to run a few iterations under each setting to obtain the data. Such overhead can be marginal as a training task usually takes hours or even up to days.

C. Proof of Theorem 1

We first introduce two lemmas.

Algorithm 1: Dynamic Programming Solution

```
Input: $K$, $Z$, $L$, $\mathbf{c}$, $\mathbf{b}$, $\mathbf{R}$
Output: tensor group size $\mathbb{X}$, thread allocation $\mathbb{Y}$
1  Initialize $\mathbf{T}[0...K][0...L][0...Z]$ with element $\infty$;
2  Initialize $\mathbf{LOG}[0...K][0...L][0...Z]$ with element $(-1,-1,-1)$;
     // Used to log $(l', z', R_j)$ for each $T[i][l][z]$
3  $w \leftarrow \frac{1}{Z}\sum_{i=1}^{L}\mathbf{b}[i]$;
4  for $z = 0$ to $Z$ do
5      $\mathbf{T}[0][0][z] \leftarrow z \cdot w$;
6  for $i = 1$ to $K, l = 1$ to $L, z = 1$ to $Z$ do
7      $t_{opt} \leftarrow \infty$, $last\_hop \leftarrow (-1,-1,-1)$;
8      $z_{min} \leftarrow$ GetMinChunkID($\mathbf{b}, l, w$);
9      for $l' = 0$ to $l, z' = z_{min}$ to $z, j = 1$ to $J$ do
10         $h \leftarrow$ GetStageTime($\sum_{i=l'+1}^{l}\mathbf{c}[i], zw, z'w, \mathbf{R}[j]$);
11         $t_{new} \leftarrow \mathbf{T}[i-1][l'][z'] + h$;
12         if $t_{new} < t_{opt}$ then
13             $t_{opt} \leftarrow t_{new}$, $last\_hop \leftarrow (l', z', \mathbf{R}[j])$;
14     $\mathbf{T}[i][l][z] \leftarrow t_{opt}$;
15     $\mathbf{LOG}[i][l][z] \leftarrow last\_hop$;
16 GatherResults($\mathbf{LOG}, \mathbb{X}, \mathbb{Y}$);
17 Function GetMinChunkID($\mathbf{b}, l, w$):
18     $id \leftarrow 1$;
19     while $id \cdot w < \sum_{i=1}^{l}\mathbf{b}[i]$ do
20         $id \leftarrow id + 1$;
21     return id;
22 Function GetStageTime($c, z_w, z'_w, r$):
23     $t_o = ..., t_b = ..., t_c = ...$; // Get the overlapped time, residual
         computation and communication time
24     if $z_w < Z \cdot w$ and $t_2 > 0$ then
25         return $\infty$;
26     else
27         return $t_o + t_b + t_c$;
28 Function GatherResults($\mathbf{LOG}, \mathbb{X}, \mathbb{Y}$):
29     // Get tensor group size and their thread allocation with
         subproblem information in $\mathbf{LOG}$
30     ...
```

Algorithm 2: LIBRA

```
Input: $K$, $Z$, $\mathbf{R}$, $model$
1  Profile $model$ on the given cluster under its training settings;
2  Collect overlap data by training $model$ with random thread
     allocation and tensor fusion policies;
3  Fit the contention model with the collected data;
4  Get tensor fusion $\mathbb{X}$ and thread allocation $\mathbb{Y}$ from Algorithm 1;
5  Run data parallel S-SGD with $\mathbb{X}$ and $\mathbb{Y}$.
```

Lemma 3. *Let $v(i, s_b^i; \phi)$ denote the time for the communication from tensor groups i to K to overlap the rest s_b^i computation under thread allocation policy ϕ. For $\forall\, \hat{s}_b^i \geq s_b^i$,*

$$v(i, \hat{s}_b^i; \phi) \geq v(i, s_b^i; \phi). \tag{24}$$

Proof. By Assumption 1, $v(i, \hat{s}_b^i; \phi)$ can be regarded as the backward time after appending $\triangle t = \hat{s}_b^i - s_b^i$ computation to the last tensor group K. By Assumption 2, the appended computation will not decrease the backward time. ☐

Lemma 4. *For $\forall \triangle t \geq 0$,*

$$v(i, s_b^i; \phi) \leq v(i, s_b^i - \triangle t; \phi) + \triangle t. \tag{25}$$

Proof. Note that $v(i, s_b^i - \triangle t; \phi) + \triangle t$ can be regarded as the time when we force the last $\triangle t$ computation to be processed without overlap. By Assumption 3, allowing the $\triangle t$ computation to overlap with communication does not increase backward time. □

Now we prove Theorem 1. Under the optimal thread allocation policy ϕ^* that achieves T^*, denote the communication start time of tensor group i as τ_i^*, and the size of the overall remaining computation at τ_i^* as s_b^{i*}. The overall backward time can be written as $T^* = \tau_i^* + v(i, s_b^{i*}; \phi^*)$. We perform $K-1$ rounds of iterative adjustment to the communication start time to make each communication start at the edge of computation chunks. We use $T^{(j)}$, $\tau_i^{(j)}$, $s_b^{i,(j)}$ to denote the new overall backward time, communication start time and remaining computation size at $\tau_i^{(j)}$ in round j, with $T^{(1)} = T^*$, $\tau_i^{(1)} = \tau_i^*$ and $s_b^{i,(1)} = s_b^{i*}$. In round j, we insert a waiting interval $\triangle t^{(j)}$ before $\tau_{j+1}^{(j)}$ to move $\tau_{j+1}^{(j)}$ to a time point corresponding to the closest computation chunk edge. The overall backward time after this adjustment can be represented as $T^{(j+1)} = \tau_{j+1}^{(j)} + \triangle\tau^{(j)} + v(j+1, s_b^{j+1,(j)} - \triangle t^{(j)}; \phi^*)$. As $\triangle t^{(j)}$ is smaller than w, we have

$$\begin{aligned} T^{(j+1)} &\leq \tau_{j+1}^{(j)} + w + v(j+1, s_b^{j+1,(j)} - \triangle t^{(j)}; \phi^*) \\ &\leq \tau_{j+1}^{(j)} + w + v(j+1, s_b^{j+1,(j)}; \phi^*) \\ &= T^{(j)} + w, \end{aligned} \tag{26}$$

where the second inequality uses Lemma 3. Thus after $K-1$ rounds of adjustment, we have $T^{(K)} \leq T^{(1)} + (K-1)w$. Since each communication starts at the edge of each computation chunk, $T(K, Z)$ is the optimal of $T^{(K)}$, and we have

$$T(K,Z) \leq T^{(K)} \leq T^* + (K-1)w = T^* + \frac{K-1}{Z}\sum_{i=1}^{K} b_i. \tag{27}$$

Recall that in practice we do not require communication to start at the edge of computation chunks; the communication of a tensor group starts immediately once its gradients are generated and communication of its preceding group has completed. The iterative adjustment above that obtains $T^{(K)}$ from T^* can be used to obtain $T(K, Z)$ from $\hat{T}(K, Z)$, so we have $\hat{T}(K, Z) \leq T(K, Z)$. Reusing the notations in previous proof, i.e. notations corresponding to T^* for $\hat{T}(K, Z)$, and those corresponding to $T^{(K)}$ for $T(K, Z)$, we have

$$\begin{aligned} T^{(j+1)} &= \tau_{j+1}^{(j)} + \triangle t^{(j)} + v(j+1, s_b^{j+1,(j)} - \triangle t^{(j)}; \phi^*) \\ &\geq \tau_{j+1}^{(j)} + v(j+1, s_b^{j+1,(j)}; \phi^*) \\ &= T^{(j)}, \end{aligned} \tag{28}$$

where the inequality uses Lemma 4. After $K-1$ rounds of adjustment, we have $T^{(K)} \geq T^{(1)}$, i.e. $T(K, Z) \geq \hat{T}(K, Z)$. As the overall backward time is no shorter than the non-overlapped backward computation time, i.e. $T^* \geq \sum_{i=1}^{K} b_i$, we have $\hat{T}(K, Z) \leq T^* + \frac{K-1}{Z}T^*$.

V. Evaluation

In this section we first validate the accuracy of our performance model given in Section III. Then we evaluate the proposed algorithm in a real-world testbed. Finally, we conduct simulation study for system scalability.

A. Methodology

Testbed. Our real-world experiment uses an 8-node GPU cluster inter-connected by an RDMA network. Each node has an Nvidia Tesla V100 GPU and a peak bandwidth of 50Gbps. The operating system is Ubuntu 18.04, and the deep learning framework is Tensorflow v2.1.0 with library CUDA-10.1 and CuDNNv7. We implement our algorithm with the distributed training framework Horovod v0.20.3 [21] and communication library NCCL v2.4.8 [23]. We use the widely adopted ring-allreduce algorithm [1] for gradient communication.

TABLE I
Models used for evaluation.

Model name	Parameter size	Batch size per GPU
Xception	$\sim$ 22M	32
ResNet152	$\sim$ 60M	32
VGG16	$\sim$ 138M	128
BERT-Base	$\sim$ 110M	32

Benchmark DNN models. We use four representative DNN models for evaluation. Xception [28], ResNet152 [29] and VGG16 [30] represent three types of the popular CNN models for image classification. BERT-Base is a transformer model for natural language processing. We train the CNN models on the ImageNet dataset [34] and BERT-Base on the BookCorpus dataset [35] with commonly used batch size. The detailed parameter size of each model is shown in Table. I.

Baselines. We compare Libra with the following baselines.

- SynEASGD (SE): SynEASGD [22] does not overlap communication and computation. It merges all gradients in a single communication task at the end of computation.
- Horovod-NCCL (H-N): Horovod-NCCL is a most widely adopted state-of-the-art distributed training framework. Horovod performs tensor fusion by dynamically grouping all gradients that are ready for communication in a predefined time window [36]. NCCL uses a set of complicated heuristic rules to allocate communication threads.
- MG-N: This baseline combines MG-WFBP [10], a state-of-the-art tensor fusion algorithm, with the communication thread allocation of NCCL.

We also compare the basic version of Libra that optimizes thread allocation for a given tensor fusion plan. We use MG-WFBP for tensor fusion and call this variant Libra-m.

Libra Settings. We set the maximum number of tensor group $K = 10$ and $Z = 150$ as we only observe marginal performance gain by further increasing the values. The set R

of allowed thread numbers includes powers of 2 from 128 to 4096. We use the same Z and R for LIBRA-M.

B. Performance Model Validation

We validate the accuracy of our performance model in Section III as follows. We first fit the model using data collected by profiling with randomly generated thread allocation and tensor fusion plans as in Algorithm 2. Then we use it to predict the backward time for training with LIBRA and the baselines in the previous subsection. Fig. 5 compares the predictions against measurements. The left four groups of points correspond to the four benchmark DNN models trained on an 8-node cluster. The prediction errors are in the range of 0.5% to 7% with an average of 2.7%, confirming the effectiveness of our model.

We also test how our model generalizes with respect to cluster size. Since it depends on the cluster size only through the startup overhead a and the transmitted data $\hat{c}$ in All-Reduce, we can easily convert the model fitted on one cluster size to a different cluster size by correcting a and $\hat{c}$ using the corresponding formulas, e.g. Eq. (4). The right two groups of points in Fig. 5 show the results for VGG16, when we apply the model obtained above on the 8-node cluster to 2-node and 4-node clusters after correction. A comparison of the three groups for VGG16 confirms that our model has good generalization ability. In Section V-D we will use the model for simulation study as we scale up the cluster size.

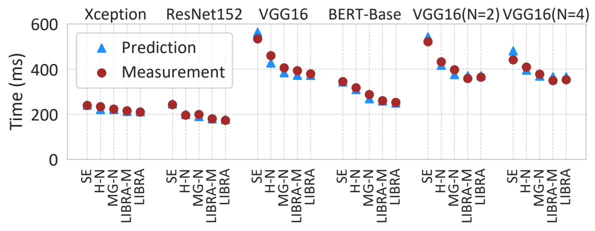

Fig. 5. Comparison between predicted and measured backward time. All models are fitted on an 8-node cluster. The fitted models are tested on the same 8-node cluster for the left four groups, and on a 2- and 4-node cluster for the right two groups, indicated by $N = 2$ and $N = 4$, respectively.

C. Real-world Experiment

Overall Performance. As the communication-computation overlap technique affects neither training convergence nor model accuracy, we focus on training throughput, i.e. the number of data samples processed per unit time, when comparing different methods. Fig. 6 shows that our method LIBRA outperforms all the baselines. The improvement is 11.3%-26.9% over SE (no overlap), 5.8%-13.8% over H-N, and 4.2%-6.2% over MG-N, the best-performing method without communication thread optimization. The variant LIBRA-M improves upon MG-N by 4.1%, demonstrating a clear benefit of incorporating our communication thread optimization into existing tensor fusion policies. LIBRA achieves additional improvement over LIBRA-M through co-optimization with tensor fusion, which suggests that the tensor fusion policy of MG-WFBP is no longer optimal in the presence of contention. Fig. 6 also shows the simulated throughput of MG-N in a contention-free environment (MG-CF), which cannot be achieve without additional resource. Note the 10.7% gap between MG-CF and MG-N due to contention. LIBRA narrows the gap to 5.5%, showing its effectiveness in mitigating contention.

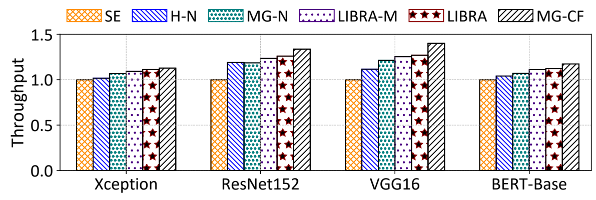

Fig. 6. Overall training performance.

Time Breakdown. Fig. 7 shows the average computation and residual, i.e. non-overlapped communication time of a training iteration. LIBRA reduces the overall backward time by 12.1%-29.1% compared with SE, by 10.1%-20.4% compared with H-N and by 5.6%-13.0% compared with MG-N. Compared with the ideal MG-CF, the backward time of MG-N is on average 22.5% higher, and LIBRA reduces the excess to 10.9%. In terms of the residual communication time, LIBRA-M and LIBRA achieve an average reduction of 11.0% and 50.1% over MG-N, respectively, without increasing computation time. Since the tensor fusion algorithm of MG-N is proved optimal in minimizing residual communication time for fixed thread allocation in a contention-free environment [10], the above observation reaffirms the importance of thread allocation optimization in the presence of contention.

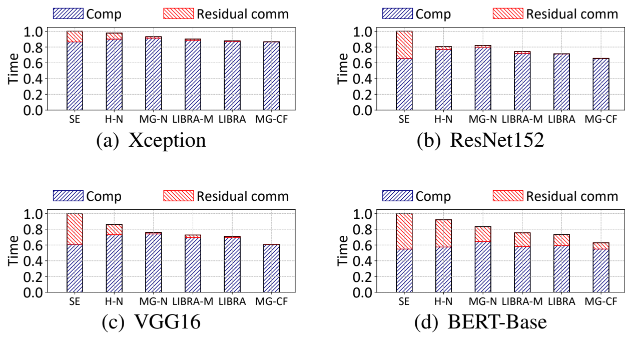

Fig. 7. Backward time breakdown.

Fig. 8 compares the computation and communication time of different methods. Note that LIBRA and LIBRA-M tend to balance the computation and communication time, which is most salient in Figs. 8(c) and 8(d). In Fig. 8(a), the computation time of LIBRA almost matches that of SE, which cannot be further reduced without additional computation resource. It is interesting to note in Fig. 8(b), LIBRA-M reduces both computation and communication time compared to MG-N, which uses the same tensor fusion plan. More detailed analysis shows that the critical path varies between computation and communication for different tensor groups, and LIBRA-M strives to reduce whichever time on the critical path.

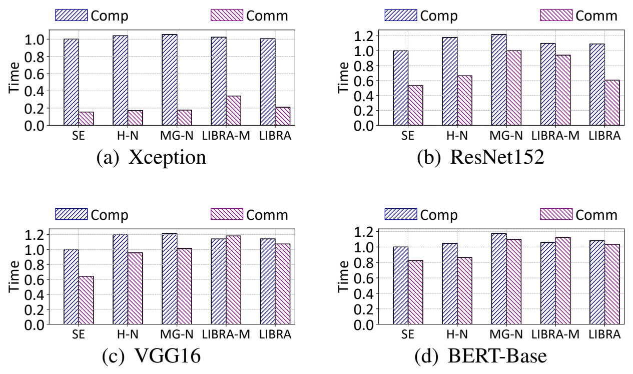

Fig. 8. Communication and computation time after overlap.

D. Simulation

Due to hardware limitation, we extend the evaluation to larger clusters by simulation based on our performance model. The generalization of a model fitted on a real 8-node cluster to other cluster sizes has been explained and validated in Sec V-B. We increase the number of nodes from 16 to 2048. Fig. 9 shows the per-node throughput, normalized by the single-node throughput. Note that the ideal linear scalability corresponds to a horizontal line with value one. While none of the methods achieve linear scalability, with all curves eventually bending down due to the increasing startup overhead in Eq. (4), LIBRA consistently outperforms the baselines. As the system scales up, the improvement of LIBRA over H-N and MG-N increases, reaching 20.0% and 21.0%, respectively, for training VGG16 on a 2048-node cluster (Fig. 9(c)). Meanwhile, the improvements of the overlap methods over SE decrease. This is because SE launches only one communication task and hence suffers less from the increase of startup overhead mentioned above.

We also observe that LIBRA-M achieves better scalability than MG-N, manifesting the benefits of thread allocation optimization. On the other hand, there is a notable gap between LIBRA-M and LIBRA when the cluster size is large (512-2048), highlighting the importance of co-optimizing thread allocation and tensor fusion when the startup overhead is large.

VI. Related Work

Comm-comp Contention Mitigation. The contention problem can be eliminated by offloading communication to a separate device [25]. However, this approach is expensive as it requires not only additional customized hardware but also the design of a new communication library. Our work provides a simple low-cost alternative to mitigate the contention problem. Our method is purely algorithmic and easy to implement.

Comm-comp Overlap Optimization. Several works focus on overlapping communication with backward computation. WFBP [9] launches a communication task as soon as gradients are ready. MG-WFBP [10] performs tensor fusion to merge short communication tasks into a single one to reduce the startup overhead. ASC-WFBP [11] launches simultaneous communication tasks to increase network utilization with TCP. None of them considers the effect of contention.

Other works further allow communication to be overlapped with forward computation through priority-based scheduling, which can be performed either at the tensor level [13–16] or at the packet level [17]. Except for ByteScheduler [14], all those works focus on the parameter-server architecture. Extending our thread allocation optimization to forward computation overlap is an interesting future work.

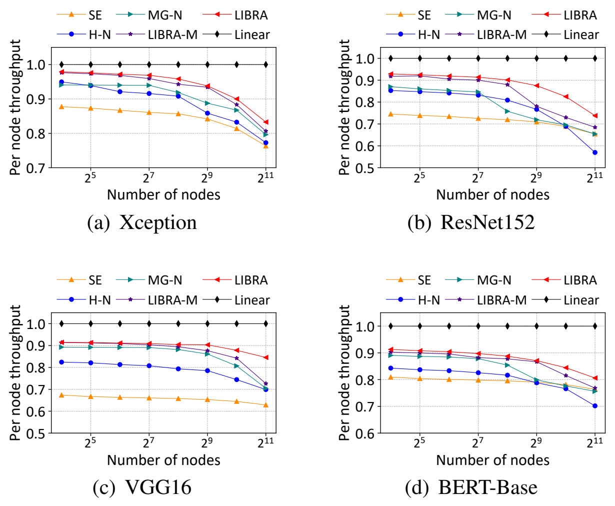

Fig. 9. Simulated per-node training throughput on clusters with 16 to 2048 nodes. Results are normalized to the throughput of training with a single node.

VII. Conclusion

In data parallel S-SGD, overlapping communication with backward computation is a popular technique to improve training throughput. However, the resource contention between computation and All-Reduce communication hinders the performance. This paper proposes a contention-aware GPU thread allocation algorithm that balances the time of overlapped computation and communication to reduce the overall backward time. We model the contention and formulate an optimization problem to decide the communication thread allocation. We develop a dynamic programming based near-optimal solution with polynomial complexity and we further extend the algorithm to co-optimize thread allocation with tensor fusion. Our real-world and simulated experiments show that our algorithm effectively mitigates the effect of contention.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China (No. 42050105, 62072302, 62272292, 61902246, 62262018, 62020106005, 62061146002, 61960206002), the Open Research Project of the State Key Laboratory of Media Convergence and Communication, Communication University of China, China (No.SKLMCC2021KF011).

References

[1] Baidu Research, "baidu-allreduce," 2017. https://github.com/baidu-research/baidu-allreduce.

[2] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[3] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *JPDC*, vol. 69, no. 2, pp. 117–124, 2009.

[4] S. Shi, Z. Tang, X. Chu, C. Liu, W. Wang, and B. Li, "A quantitative survey of communication optimizations in distributed deep learning," *IEEE Network*, vol. 35, no. 3, pp. 230–237, 2020.

[5] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed GPU with a GPU-specialized parameter server," in *Proc. of EuroSys*, pp. 1–16, 2016.

[6] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: a rack-scale parameter server for distributed deep neural network training," in *Proc. of SoCC*, pp. 41–54, 2018.

[7] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun, "Parallax: Sparsity-aware data parallel training of deep neural networks," in *Proc. of the Fourteenth EuroSys*, pp. 1–15, 2019.

[8] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," *VLDB Endow.*, 2020.

[9] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *ATC*, pp. 181–193, 2017.

[10] S. Shi, X. Chu, and B. Li, "MG-WFBP: Merging gradients wisely for efficient communication in distributed deep learning," *IEEE TPDS*, vol. 32, no. 8, pp. 1903–1917, 2021.

[11] S. Shi, X. Chu, and B. Li, "Exploiting simultaneous communications to accelerate data parallel distributed deep learning," in *Proc. of INFOCOM*, pp. 1–10, 2021.

[12] S. Shi, Q. Wang, X. Chu, B. Li, Y. Qin, R. Liu, and X. Zhao, "Communication-efficient distributed deep learning with merged gradient sparsification on GPUs," in *Proc. of IEEE INFOCOM*, pp. 406–415, 2020.

[13] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *Proc. of IEEE INFOCOM*, pp. 626–635, 2020.

[14] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *SOSP*, pp. 16–29, 2019.

[15] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," *MLSys*, pp. 132–145, 2019.

[16] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," *MLSys*, pp. 418–430, 2019.

[17] Q. Duan, Z. Wang, Y. Xu, S. Liu, and J. Wu, "Mercury: A simple transport layer scheduler to accelerate distributed DNN training," in *INFOCOM*, pp. 350–359, 2022.

[18] Y. Yu, J. Wu, and L. Huang, "Double quantization for communication-efficient distributed optimization," *NIPS*, vol. 32, 2019.

[19] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," *NIPS*, vol. 31, 2018.

[20] T. Vogels, S. P. Karimireddy, and M. Jaggi, "Powersgd: Practical low-rank gradient compression for distributed optimization," *NIPS*, vol. 32, 2019.

[21] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in Tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[22] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on GPU and knights landing clusters," in *SC*, pp. 1–12, 2017.

[23] Nvidia, "Nvidia collective communications library (NCCL)." https://developer.nvidia.com/nccl.

[24] Intel, "oneapi collective communications library (oneccl)." https://github.com/oneapi-src/oneCC.

[25] S. Rashidi, M. Denton, S. Sridharan, S. Srinivasan, A. Suresh, J. Nie, and T. Krishna, "Enabling compute-communication overlap in distributed deep learning training platforms," in *Proc. of ISCA*, pp. 540–553, 2021.

[26] G. Gilman, S. S. Ogden, T. Guo, and R. J. Walls, "Demystifying the placement policies of the NVIDIA GPU thread block scheduler for concurrent kernels," *ACM SIGMETRICS Performance Evaluation Review*, vol. 48, no. 3, pp. 81–88, 2021.

[27] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *RTSS*, pp. 104–115, 2017.

[28] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *CVPR*, pp. 1251–1258, 2017.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, pp. 770–778, 2016.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2018.

[32] Nvidia, "GEFORCE GTX 1080 Ti." https://www.nvidia.cn/geforce/products/10series/geforce-gtx-1080-ti/.

[33] Nvidia, "NVIDIA TESLA V100." https://www.nvidia.com/en-gb/data-center/tesla-v100/.

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, pp. 248–255, 2009.

[35] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *ICCV*, 2015.

[36] J. Romero, J. Yin, N. Laanait, B. Xie, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Accelerating collective communication in data parallel training across deep learning frameworks," in *NSDI*, pp. 1027–1040, 2022.